University of Zurich
Department of Informatics

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Knowledge Management and Sharing in the Framework of Project SciMantic

Daniel Spicar
Zurich, Switzerland
Student ID: 06-702-542

Supervisor: Dr. Hasan
Date of Submission: October 6, 2010

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Einleitung

Diese Arbeit befasst sich mit Informatiksystemen, welche die Verwaltung und das Teilen von Wissen unterstützen. Es werden viele Daten generiert ohne dass darin enthaltenes Wissen systematisch verwaltet wird. Dadurch kommt es oft vor, dass Arbeiten mehrmals erledigt werden. Abhilfe können Applikationen schaffen, die Wissen so aufbereiten, dass es leicht zu finden und zu verwenden ist. In dieser Arbeit wird der Schwerpunkt auf die Kombination von Semantic Web und Peer-to-Peer Technologien gelegt. Mit Semantic Web Technologie kann man Dinge so annotieren, dass auch Maschinen deren Bedeutung verstehen können. Das eignet sich hervorragend für die Verwaltung von Wissen, da man Informationen zusammen mit Ihrer Bedeutung transportieren kann. Peer-to-Peer Technologie erlaubt die gemeinsame, effiziente Nutzung verteilter Ressourcen. Diese Arbeit verwendet verteilte Hash-Tabellen (DHT) um Wissen mit anderen Applikationen zu teilen.

Ziele

Das Resultat dieser Arbeit soll eine Architektur und einen Prototyp eines Knowledge Sharing Systems (Wissensteilungssystems) sein. Dieser Prototyp muss mindestens folgende Funktionalität aufweisen:

- Architektur und Implementierung eines Verwaltungsdienstes mit einer Web Schnittstelle, die die Erzeugung, Änderung, Löschung und das Lesen von Wissenseinheiten erlaubt.
- Architektur und Implementierung eines Dienstes, der einen verteilten Index von Wissenseinheiten erstellt und pflegt. Der Dienst muss zusätzlich Stichwortsuche nach Wissenseinheiten erlauben.
- Architektur und optionale Implementation eines Dienstes der das Teilen von Wissenseinheiten mit anderen Applikationsinstanzen erlaubt.
- Leistungs- und Skalierbarkeitsevaluierung der implementierten Lösung.

Zusätzlich soll die Arbeit die Vorteile und Nachteile der Kombination von Peer-to-Peer Technologie mit Semantic Web aufzeigen.

Resultate

Es konnte aufgezeigt werden, dass eine Kombination der beiden Technologien nicht nur möglich ist, sondern auch skalierbar und efficient sein kann. Es gibt Situationen, in denen die Lösung aber nicht gut funktioniert. Zum Beispiel für verteilte RDF Speicher. Ohne starke Optimierungsbemühungen ist der Speicher langsam. Momentan ist der Speicher speziell auf Wissenseinheiten optimiert. Dies macht die Lösung effizient aber dafür ist der Inhalt der Wissenseinheiten nicht semantisch durchsuchbar ohne die ganze Wissenseinheit herunterzuladen. Andererseits kann man am Beispiel der Stichwortsuche die Vorteile der Kombination beider Technologien erkennen. Anstatt nur nach Text zu suchen, kann man mit Hilfe von RDF die Bedeutung der Stichworte erkannt und berücksichtigt werden. Damit lassen sich bessere Suchergebnisse erzeugen und logische Inferenz auf den Daten betreiben.

Ausblick

Der implementierte Prototyp hat noch einige Mängel. Vor allem muss der DHT-Dienst optimiert werden. Dann könnte man auch einen echten verteilten RDF Speicher mit den nötigen Optimierungen implementieren. Zudem mangelt es in der aktuellen Implementation an Sicherheitsmechanismen für den DHT Zugriff. Jeder kann in der DHT schreiben und lesen. Und nicht zuletzt braucht es ein grafische Benutzoberfläche für menschliche Benutzer. Bisher muss der Prototyp mit Konsolenbefehlen bedient werden.

Abstract

In a time when data is abundant and collaboration is in the center of research and business activities it is important to turn information contained in data into knowledge and make that knowledge available to people. The intention is to *avoid reinventing the wheel*. This can be done with knowledge management. Knowledge management systems are information systems that support people in knowledge-based activities.

With the semantic web, a technology becomes widely available that can transport not only data but also meaning. Semantically annotated data is accessible to machines as well as humans. It is an ideal technology to build knowledge management systems with. Peer-to-Peer technology on the other hand specialises on scalable and efficient sharing of resources.

The SciMantic projects wants to provide a semantic web application that focuses on knowledge sharing. This thesis explores the advantages and disadvantages of the combination of semantic web and Peer-to-Peer technologies. It designs an architecture and provides a prototypical implementation. Furthermore it can demonstrate that the two technologies can be combined to build a scalable knowledge sharing system that can benefit from semantic web technologies. But it also shows that distributed RDF stores are not very efficient and need much optimization before they can be used in real life systems.

Acknowledgments

I would like to thank the Communication Systems Group at the Department of Informatics of the University of Zurich, especially Prof. Burkhard Stiller for providing me with the opportunity to write this thesis.

I would like to thank Dr. Hasan for supervising my work and providing me with valuable advice and support.

Special thanks go to Věra for all the support and proof reading.

Contents

Chapter 1

Introduction

The semantic web refers to technologies that aim to realize a web of data. The vision is that data can be linked together just like documents are linked together in the traditional World Wide Web. If this is done in a formal manner it can help agents to navigate distributed data and derive meaning from it. Then not only humans, but also machines, can understand the semantics of resources on the web. This vision is driven by the World Wide Web Consortium (W3C) that has defined several key technologies for the semantic web. Most notably perhaps, the Resource Description Framework (RDF) [19] that provides the foundation for linking and publishing data on the web. But today the question of how to link RDF data sources distributed over different sources is not yet fully decided.

Peer-to-Peer (P2P) systems are one possible approach of sharing distributed resources. Today P2P Applications are used mostly for file sharing but other ways of using them exist as well. P2P networks are decentralized systems that have proven themselves reliable, because they have no single point of failure and they exhibit a degree of self-organisation that allows them to adapt to a dynamic topologies.

The SciMantic project [21] explores whether it is beneficial to use P2P networks to build distributed semantic web applications. For this purpose a collaboration between Trialox and the University of Zurich has been established. Trialox is the company behind the Apache Clerezza [15] semantic web platform and both, semantic web and Peer-to-Peer systems, are a research interest at the University of Zurich. SciMantic intends to provide a Semantic Content Infrastructure (SCI) that enables applications built on top of it to efficiently work with semantic content shared among independent instances of these applications. This capability should be attained by combining semantic web and P2P technologies. In [11] a Knowledge Sharing System (KSS) is defined as a possible application scenario. A KSS is an application that allows users to share knowledge within Knowledge Sharing Networks (KSN) specialising on different topics, such as medicine, computer science, or politics. Humans and machines can interact with the KSS trough web services. Figure 1.1 shows the concept of the SciMantic KSS.

This thesis focuses on a prototypical implementation of a Knowledge Sharing System as a SciMantic web application on top of the Apache Clerezza platform. Clerezza is an

OSGi [26] based platform and provides tools to build RESTful web applications. OSGi is a framework that facilitates modular programming and dynamic class loading in Java. Reduced complexity, facilitated reuse of exiting modules, and a simple framework for service oriented architectures are only three of the benefits of using OSGi. REST stands for Representational State Transfer [14] and is a software architecture style. REST compliant architectures are called RESTful. In the context of web services REST distinguishes itself by maximizing the reuse of the well defined, general interfaces of the HTTP protocol. It is easier to use RESTful web services as a user does not need to familiarize himself with new protocols and interfaces for every application.

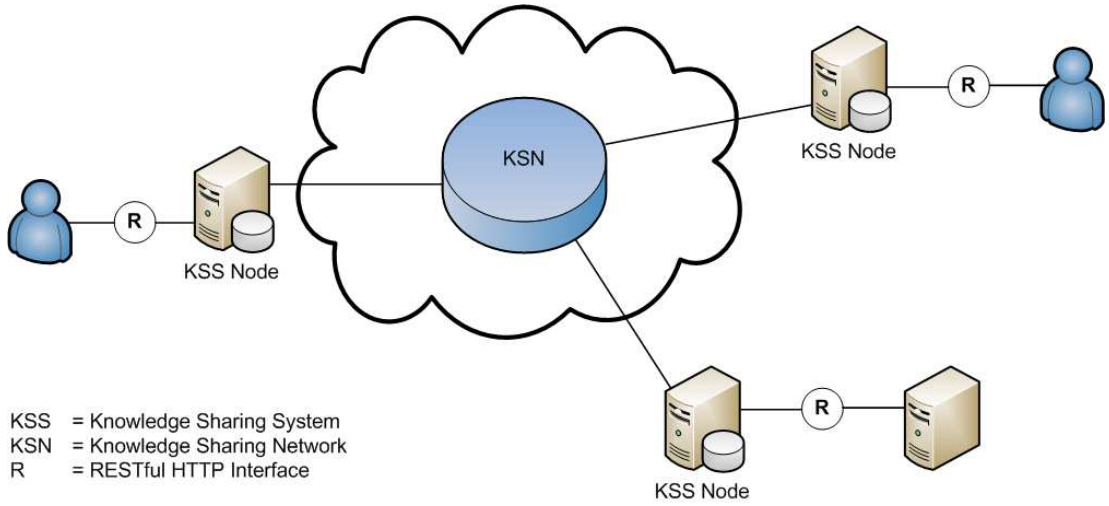

Figure 1.1: SciMantic KSS concept: Users interact with the system trough web interfaces. Single instances of the KSS connect to Knowledge Sharing Networks in order to share resources.

1.1 Motivation

Information technology enables people to access and create vast amounts of data. In this data knowledge is contained and it represents a valuable resource. The task of knowledge management is to extract knowledge from data and then manage it. Knowledge about situations, contexts, or processes guides the way many organisations work these days. If knowledge is not managed, there is a risk of doing unnecessary, redundant work which can consume considerable resources [18]. The key goal is that knowledge that has been attained in the past, must be accessible so it can be reused in the future. Knowledge management tries to enrich and organize knowledge in such a way that it can be used in future processes.

Information technology offers an approach to knowledge management. Applications that support sharing and finding knowledge in repositories can help to make it accessible and therefore useful. This thesis focuses on the implementation of an application that does this. In section 2 systems that have a wider scope and encompass knowledge extraction and enhancement are presented as well.

While knowledge management and sharing establish the context of this thesis, it concentrates on technological research and explores the benefits of the combination of P2P and semantic web technology. If the semantic web vision is to come true, the distributed sources of RDF data on the Web need to be connected. By implementing an application that does this we hope to demonstrate advantages and disadvantages of our approach.

Alternative approaches follow a pattern of centralized indices or services. Google has been very successful in indexing the World Wide Web in this manner and there is no reason why it should not be possible to index the web of data in the same manner. But the amount of resources required to build and maintain indices of the World Wide Web is immense. Centralized indices require resources to be centralized as well. But the internet has been designed as a decentralized system and we believe that a distributed solution makes better use of the available resources and lowers the obstacles for everyone to participate in realizing the web of data.

1.2 Description of Work

A subset of functionalities defined for SCI are implemented in Apache Clerezza. Therefore this software platform should be used to build a KSS. Knowledge within the KSS is divided into knowledge units. A KSS consists of three services [11]:

Knowledge Unit Sharing Service (KUSS) This service allows users to publish their knowledge units. It also provides a subscription service that notifies users of changes defined by user criteria.

Knowledge Unit Management Service (KUMS) This service provides for storage of knowledge units and management operations including create, read, update, and delete (CRUD).

Knowledge Unit Tagging and Searching Service (KUTSS) This service allows users to tag a knowledge unit with appropriate keywords and to search for knowledge units tagged with specific keywords. Searching of knowledge units is not restricted to local storage.

The detailed tasks in this thesis are:

- Design of an architecture providing the three KSS services.
- Design and implementation of OSGi bundles providing the knowledge unit management service.
- Design and implementation of OSGi bundles for distributed indexing and keyword-based search for knowledge units.
- Design of bundles providing the knowledge unit sharing service. Its implementation is optional.

- Performance evaluation of the implemented services.

The results of this thesis are to to show the advantages and disadvantages of the combination of semantic web and P2P technologies, particularly in developing a system providing knowledge sharing trough web services running on each peer. Furthermore a scalability evaluation in terms of number of peers and number of knowledge units and a performance evaluation of the implemented system shall be provided.

1.3 Thesis Outline

In chapter 2 related knowledge management systems are presented and compared to the SciMantic knowledge sharing system. Chapter 3 introduces the reader to knowledge management and knowledge management systems in general. Then the Resource Description Framework is explained to illustrate the fundamentals of the semantic web. A look at Peer-to-Peer systems and distributed RDF stores completes the introduction to these key technologies that are the foundation of this thesis. In chapter 4 the concepts of the SciMantic knowledge sharing system are described and an architecture of the system is designed. Chapter 5 describes the implementation of the system, the difficulties encountered, and their resolution. In chapter chapter 6 the implemented system is evaluated with respect to performance and scalability and the results are summarized in chapter 7 before the thesis concludes with an outlook on further work.

Chapter 2

Related Work

2.1 K-Net – Services for context sensitive enhancing of knowledge in networked enterprises

The K-Net research project is participating of the European Union's 7th Framework Programme (FP7). FP7 is the EU's main instrument for funding research [3]. K-Net focuses on three main aspects [4].

- Knowledge Monitoring: Collaboration support services generate information by monitoring knowledge-based activities, e.g. team meetings.
- Context Extraction: The context in which knowledge-based activities take place contains valuable information. K-Net provides services which extract context information.
- Knowledge Enhancing: In order to maximise the reuse potential of stored knowledge, a service enhances it with context information. This enables context-sensitive knowledge provisioning.

These services are aimed at networked, small and mid-sized enterprises and addresses partners from an industrial environment as well as the research and technology development community.

2.1.1 K-Net Scope

The K-Net project researches methodologies and approaches to knowledge sharing and management and develops specifications and implementations of the required services. Their core research involves the question of how to store and exploit implicit information given by the context of knowledge-based activities.

> "The key hypothesis of the project is that the context under which the knowledge is collectively generated and managed can be used to enhance this knowledge for its further utilisation within the intra-enterprise collaboration." [12]

2.1.2 K-Net Technology

In order to extract implicit information from knowledge-based activities, services need to understand their context. For this purpose a model has been created. This model, the K-Net Ontology, uses Semantic Web technologies such as the RDF data model, the OWL web ontology language and SKOS to annotate knowledge items and enable reasoning on the semantic data to take place [12].

The K-Net project uses the Jena Framework [1] ref for building Semantic Web applications to implement the necessary services. Jena includes a triple store (persistent storage solution for RDF data) and APIs for working with Semantic Web standards such as RDF, OWL, and the SPARQL RDF query language [22]. Web services are implemented using the SOAP protocol and interfaces are described with the Web Service Description Language (WSDL). Business logic is written as Enterprise JavaBeans (EJB) placed on a JBoss Application Server. Aside from the Java Programming language all tools are open source software, and all formats used are open standards. [10]

2.2 SharK – Shared Knowledge

Shark is a framework for building distributed, context-based information systems [24]. Information can be of any format or type and is always assigned to a context. SharK is based on an unstructured P2P network and uses a custom P2P protocol called KEP (Knowledge Exchange Protocol). KEP is designed for knowledge exchange between peers based on interests specified by the participating peers. In SharK knowledge received from other Peers is not trusted by default and needs to be assimilated into the local knowledge base before it can be used. During that process the local knowledge base is extended and can generate new information from the new combined knowledge (inference). KEP peers are assumed to be mobile and can *meeet* in the same physical location to exchange knowledge. Instead of the Semantic Web approach SharK uses *topic maps* to link information with its context and assign meaning to the information in that way.

2.3 Comparison to SciMantic

From these two example we see that both of these related systems share elements with the SciMantic KSS (SKSS). But each of them has unique features as well. The focus of the SKSS is primarily on knowledge sharing. K-Net focusses on extracting the context

[1] http://jena.sourceforge.net/

	Clerezza	**K-Net**	**SharK**
Focus	Sharing	Context Extraction	Exchange/Sharing
Com. Paradigm	P2P/DHT	C/S	P2P/random
K. Representation	RDF	RDF	Topic Map
philosophy	open sharing	controlled network	mobile devices

Table 2.1: Comparison of SciMantic KSS, K-Net, and SharK

information form knowledge-based activities. SharK focuses on knowledge exchange which can be considered knowledge sharing as well. But it does not exchange identical knowledge between peers. Instead it offers knowledge to peers who can then interpret it and possibly enrich their own knowledge base. After this process the knowledge item on the sender does not have to be identical to the knowledge item on the receiver.

In terms of communication paradigms, both SKSS and SharK use P2P approaches. SKSS uses distributed hash tables (DHT) and SharK uses a pure unstructured P2P networks. K-Net only uses client/server communication.

Knowledge representation in K-Net and SKSS is done using RDF and other Semantic Web technology. SharK uses a concept called Topology Map.

The audiences and areas where the systems are to be used differ for all three systems. SKSS follows an open sharing approach. It means knowledge can be shared among everyone interested in it. K-Net focusses more on businesses that usually have closed, controlled networks and are not interested in sharing their knowledge with the outside world. SharK envisions mobile peers that can meet anywhere to exchange knowledge. SharK defines business scenarios as well but the idea of physically mobile Peers is unique to it.

Table 2.1 summarized the discussion in this section.

Chapter 3

Background

This section provides the reader with an overview of selected topics and technologies that are required to understand the context of this thesis and the concepts presented in section 4.

3.1 Knowledge Management and Knowledge Management Systems

In order to successfully collaborate, discrepancies between the knowledge of individuals have to be overcome and a common knowledge-base must be established. This process often happens naturally in small scale collaborations where people work together and exchange information and opinions in meetings. However in large organisations it may not always be feasible for all people to be in direct contact and separate groups may arrive at very similar problems independently of each other. How can everyone still profit from relevant knowledge that exists in the organisation to solve the problems more efficiently? Additionally people involved in processes within the organisation change over time and when some experts are not available any more, their knowledge may be lost to the organisation. This can have severe consequences for businesses that face harsh competition [5]. In order for organisations to be less dependent on individuals and their knowledge and to facilitate large scale collaboration and knowledge sharing, knowledge has to be managed and shared. This can be done with processes, activities and technological tools.

The relationship between data/information and knowledge is ambivalent. A commonly held view is that information becomes knowledge when it is processed in the mind of individuals and knowledge becomes information when it is represented in symbolic form (text, graphics, etc.). As a consequence of this view we can expect that knowledge management systems are in essence normal information systems [5]. They enable users to assign meaning to information in order to build a common knowledge-base among all actors (people and machines) interacting with the system. Only a common knowledge-base enables all actors to arrive at the same understanding of data.

This thesis focuses on knowledge management and sharing from a techno-centric point of view. For this purpose a knowledge management and sharing application is being developed.

3.2 The Resource Description Framework – RDF

As a Semantic Web application, the KSS uses the Resource Description. This section provides an overview of the key elements of RDF as defined by the World Wide Web Consortium (W3C) [19].

RDF is a language to represent information about resources on the World Wide Web. The concept of such resources is very general. Anything that can be identified on the Web can be described with RDF. A straight forward example is an image, which has meta information that can be described with RDF. But it can also be a person, such as the author of a web page, who is described in terms of his name, e-mail address, and phone number. Things in RDF are identified using Uniform Resource Identifier (URI) references. A URI is a more general concept than a Uniform Resource Locator (URL). URLs are used to identify network locations. As with the example of the author of a web page, URIs can identify things that are not necessarily network locations.

To describe a resource with RDF it is first necessary to assign an URI to the resource. A resource can be described with properties that have values. Such as a person (resource) having a name (property) which is "John" (value), see figure 3.1. In a linguistic interpretation we are making statements about the resource. The resource is the subject of the sentence, the property is a predicate, and the predicate's value is the object. Because each statement consists of three parts in RDF, these statements are also called triples.

Subjects and properties are described with URIs. Anonymous subjects, termed blank nodes, also exist. They have no identifier and are useful in special cases only. Most subjects have to be uniquely identified and require URIs. But why is a general concept such as a property that serves to describe a person's name assigned a URI? When a property is assigned, an interpretation of this property is assigned with it. Using URIs enables a specific interpretation of a property to be identified. In the example above: Let the name be a one word literal (can not contain spaces) that is interpreted as user name of an application. Somebody else can define that the name of a person must be the person's full name. Both interpretations are incompatible and therefore unique identification of the property is necessary to derive the correct meaning. Objects on the other hand can be either URIs, blank nodes, or literals. Literals can have various data types but are represented as strings.

Collections of RDF statements describe directed multi-graphs where predicates are arc between two nodes. Arcs originate at the subject and pointing to the object. Figure 3.1 shows an RDF graph of the example above. A resource can have any number of outgoing and incoming arcs.

As an open standard RDF enables the exchange of information between applications that have been created independently of each other. By the use of URIs ontologies that specify

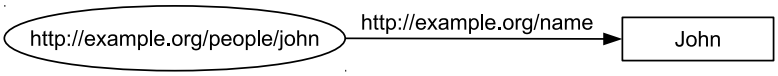

Figure 3.1: An example of an RDF graph.

the semantics of resources and properties can be created. Therefore RDF can not only transfer information but also meaning. Ontologies can be described with RDF itself using the web ontology language (OWL) specified by the W3C [16]. Furthermore OWL enables applications to perform logical reasoning on RDF data.

3.3 Peer-to-Peer Networks

Peers (also referred to as nodes) are the entities that form Peer-to-Peer networks. Peers act as clients and servers at the same time and they share the same set of capabilities. Each peer is able to perform the same roles in a P2P network.

P2P networks are formed by direct connections between peers. Peers join these networks using a mechanism called bootstrapping. Bootstrapping is required because the network is not formed around a central entity. Therefore a node that wants to join an existing P2P network needs to acquire the address of a peer that already participates in the network. By contacting this peer the node can start a join procedure. This address acquisition and joining procedure is referred to as bootstrapping.

The defining characteristics of unstructured P2P networks are that they operate without central control in a self-organizing manner to enable all peers to share resources among each other. P2P systems have a special resilience against node failure because there is no central point that can compromise the entire network. They also deal very well in a dynamic environment where nodes join and leave at all times due to their self-organizing nature. P2P networks form so-called overlay networks because they exist on top of other networks, such as the internet. This means that P2P networks use the internet as a transport network. The network topology of the overlay network does not correspond to the topology of transport network.

Content search in unstructured P2P networks is done using flooding. A node that is looking for content is broadcasting the search query to all its neighbours. If the neighbours can not answer the query, they broadcast the query to all their neighbours. This has the advantage that queries can be complex and contain wild cards but it creates a lot of unnecessary network traffic because most likely the majority of nodes can not answer the query. This can only be kept within limits by limiting the number of hops a query can travel. This limitation also means that it can not be guaranteed that a result is found when one exists.

There are various forms of P2P networks that do not follow all of these principles. Some systems create hierarchical structures with special classes of peers. Usually such hybrid systems are employed to optimize the performance of P2P networks for content lookup.

Another variation are structured P2P networks, where all P2P principles are followed but special attention is given to how nodes are interconnected and what kind of routing information they keep to facilitate content lookup. These systems are also known as Distributed Hash Tables (DHT). Distributed Hash Tables perform efficient, reliable, and scalable content retrieval with exact match queries. Exact match queries means that fuzzy queries involving wild cards are not supported. These system apply a hash function on the query and search for the exact resulting hash key.

DHTs can be best imagined as a ring structure. The address space of the nodes is mapped on this ring. Nodes are located at specific points around the ring, each of them has a predecessor and a successor. All nodes in the ring maintain some sort of routing information about their neighbourhood. When content is saved into the DHT, a hash function is applied. The resulting hash key is mapped on the node address space. Conceptually this means that the value is assigned an address in the ring where it is to be saved. But that address does not necessary have a node assigned to it. Various schemes exist on how to handle this common situation but a straight forward approach is, to save it at the closest successor node. Searches are equivalent with routing to a specific address. Searches can start at any node. The key is hashed and then the request is routed along the ring until the node responsible for the key is found. Routing performance can be optimized at the cost of higher maintenance effort. On average DHTs achieve logarithmic content lookup performance and they always find a result to a query when one exists.

3.4 Distributed Triple Stores

In order to realize the Semantic Web's vision of linked data, RDF data distributed over the web needs to be connected. Centralized triple stores (solutions for persistent RDF triple storage) such as Jena TDB [1] or Mulgara [2] are getting reliable, fast and popular. But to harness the true power of the Semantic Web remote triple stores need to be queried as well. One approach is to provide web interfaces where search queries can be performed on those triple stores (for example SPARQL [22] end points). This follows a classical client/server paradigm and implies that the servers offering the query interfaces need to deal with high load when their data sets are very popular. To deal with high load powerful hardware and mechanisms to spread queries over multiple servers for parallel processing are needed. Some centralized triple stores are available as clustered editions for this purpose. But we believe that P2P technology offers a more reliable solution that shares available resources more efficiently. In the next section a solution for a distributed triple store that makes use of DHTs is examined.

3.4.1 RDFPeers – A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network

RDFPeers proposed by Cai and Frank [8] is a distributed triple store. Triples are saved in a Chord-based DHT by applying a hash function to their subjects, predicates, and objects. The triple is saved under each of the resulting keys. This means triples are saved three times. With this solution load balancing issues appear on nodes that are tasked to save triples with very common predicates like *rdf:type* or, depending on application, very common object literals. RDFPeers resolves this problem by making nodes refuse to save triples that are saved under too common subjects, predicates, or objects. This means that when a node saves triples based on the predicate *rdf:type* and the number of saved triples reaches a defined threshold, it will not save any more triples based on this predicate. The rationale behind this behaviour is that queries for too frequent keys, such as *rdf:type*, are too general anyway to give meaningful results. Instead for searching for a triple based on *rdf:type* it is better to search based on its subject or object. A search for *rdf:type* returns too many unrelated triples because *rdf:type* is a very common predicate that most resources have even though they may have nothing in common.

For searching RDFPeers supports a native query language that can answer most basic queries in log(N) routing hops, with N being the number of nodes in the DHT. Additionally range queries are supported too. Furthermore there is a processor to translate RDQL [25] queries into native queries.

Chapter 4

Concept

The following sections provide the detailed concept of an SCI-based Knowledge Sharing System and its components in the framework of project SciMantic. The focus is on abstract concepts, architectural elements, and technology that may be used to realize them. Furthermore design choices are justified. The goal is that the reader can imagine how the KSS application is structured and how it is intended to work. Note that in this section a theoretical model is created. The concrete implementation may deviate from it.

4.1 Knowledge Units

Knowledge units (KU) are the smallest globally identifiable pieces of knowledge in the scope of the KSS. A knowledge unit can be a complex conglomerate of resources and references but to the knowledge sharing system it is an inseparable unit. To illustrate this on an example let us assume a knowledge unit to be a scientific report. But it can also be any other type of information such as an audio file, video file, or a web page. A report contains among other things: structured text, formatting information, figures, a list of keywords, references, and meta information. The KSS treats all this information as a unit and assigns a globally unique identifier to each knowledge unit that is entered into its system.

4.1.1 Keyword Tagging

In order to enhance searching possibilities a service that allows knowledge units to be tagged with keywords is provided. The tagging can be done by user interaction or by a dedicated application. The details of the tagging service are beyond the scope of this thesis. Therefore all knowledge units are assumed to be tagged by appropriate means already. To make use of the keyword tags, a dedicated searching service builds a distributed keyword index and allows users to search for knowledge units with keywords. Without a keyword-based search mechanism a user has to know the identifier of a knowledge unit

in order to access it. An identifier is long and possibly cryptic and therefore does not represent an appropriate mean for user interaction.

In order to bridge language differences and to enable extended reasoning, keywords are modelled as resources representing a concept and they receive a URI in the SciMantic namespace. Then labels (the actual keywords) can be added in different languages and alternative spellings. This also avoids confusion between keywords that are spelled the same but refer to different concepts (such as apple fruits and Apple computers). Additionally concepts can be interlinked and describe their relations to each other. This may be used in the future to provide not only keyword search but to find related knowledge units using the context of keyword concepts. This gives an impression of how this application can benefit from semantic web technologies. The W3C defined the Simple Knowledge Organisation System (SKOS) [6] vocabulary to describe such concepts and their relations in RDF.

4.1.2 Knowledge Unit Ontology

Knowledge units are described as RDF graphs and SciMantic aims at reusing open vocabularies and ontologies. Open standards reduce the barriers for third party applications to process and reason about SciMantic knowledge units. Examples of the vocabularies and ontologies used are those defined by the Dublin Core Metadata Initiative (DCMI) [7] to describe general metadata and SKOS for keyword tagging.

The content of a knowledge unit is described in the DiscoBit Ontology [1] because Clerezza provides an inbuilt DiscoBit editor. But the KSS does not process the internals of a knowledge unit beyond the metadata that it requires for indexing and administrative management. The tasks it fulfils in respect to knowledge units are:

- Extraction of knowledge units from submitted RDF graphs.
- Notification of users about new and updated content.
- Sharing and deleting knowledge units in a KSN.
- Authentication of write access to knowledge units.
- Indexing of knowledge units to provide for keyword-based search.

Given these requirements the following resources and predicates are needed. The numbers correspond with the numbers in figure 4.1.

- a knowledge unit resource (1).
- a knowledge unit RDF type (2).
- Date of submission and last modification of a knowledge unit (4, 5).

[1] `http://discobits.org/`

- An identification of the KSN a knowledge unit is shared in (property is missing if the KU is not shared) (6).
- The owner of a knowledge unit (3).
- Keyword tags with assigned weight denoting their importance or relevance (7, 8).

The *kss* namespace (as seen in figure 4.1) is defined by a custom SciMantic ontology defined specifically for the KSS.

The KSS is indifferent towards the rest of the knowledge unit graph. If the KSS can identify a (sub-)graph as a knowledge unit using the *rdf:type* property, it extracts the entire knowledge unit using a graph traversal algorithm.

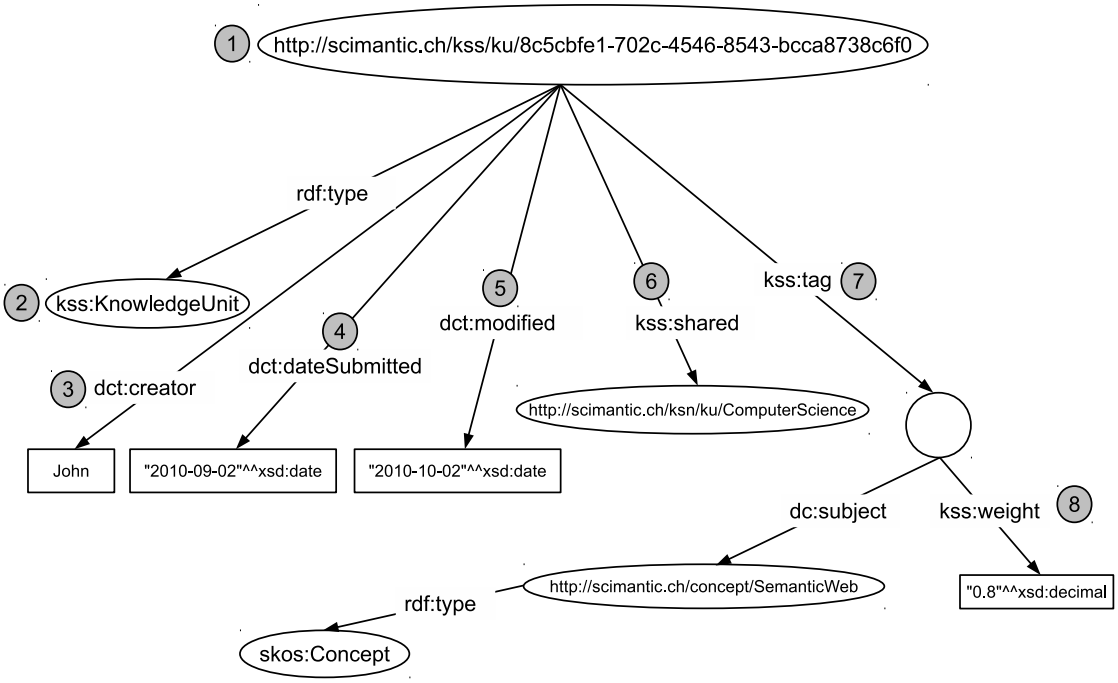

Figure 4.1: Parts of a RDF Graph describing a knowledge unit.

4.2 Knowledge Sharing Networks

Knowledge Sharing Networks connect KSS instances for the purpose of sharing knowledge units of a specific topic or interest. As described in section 1, these interests can be scientific disciplines such as medicine. But no restrictions on defining the scope of a KSN are specified. KSNs can and should form with any scope when there is a requirement for them. A single KSS instance can join any number of KSNs. A user can also decide to leave a KSN again. In that case the shared knowledge units are removed from the KSN. It is important to note that joining and leaving KSNs does not correspond with the KSS instance being operational or shut down. Joining and leaving are conscious decisions

by the KSS user. Even when a KSS temporarily drops out of the network the shared knowledge units should remain available.

Triple stores are solutions to persistently store RDF statements (so-called triples, see sections 3.2 and 3.4). They provide for efficient storage and retrieval of RDF triples and often offer advanced query language processors. Knowledge units are collections of triples that can be saved in KSNs. Conceptually a KSN can be seen as a distributed triple store.

SciMantic uses P2P technology to connect KSS instances. Therefore KSS instances are modelled as nodes participating in a P2P overlay network. The only infrastructure a Knowledge Sharing Network must provide is a bootstrapping mechanism that allows new nodes to join an existing network. All the other mechanisms are provided for by the participating nodes. These mechanisms and bootstrapping are described in more detail in section 3.3.

4.3 Distributed Indexing of Knowledge Units

Section 4.1.1 demonstrates why knowledge units are annotated with keywords. Distributed triple stores that support complex queries could solve the keyword search problem. But because anonymous blank nodes are used for knowledge unit tags (c.f. 4.1) these queries are non-trivial. Keyword search is the preferred method of searching knowledge units in the SciMantic KSS and a dedicated distributed index can help to optimise search performance. The following sections describe a basic solution where an index is saved in a DHT. Later two more advanced solutions are presented.

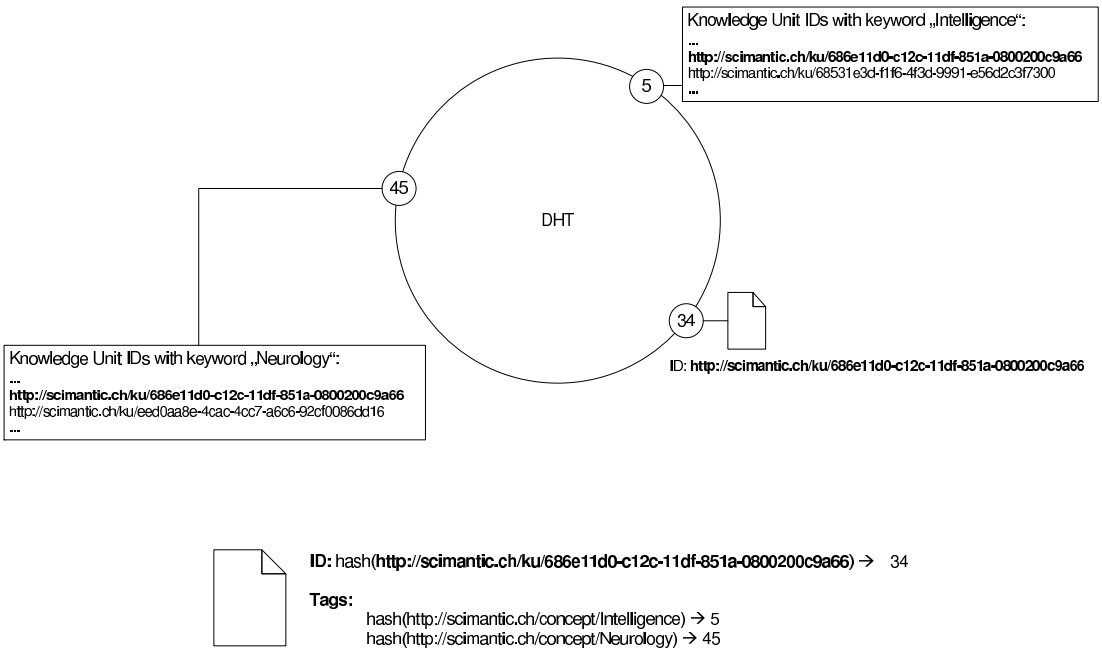

Figure 4.2: DHT Organisation: Knowledge units are saved according to their URI. Additionally the hash of keyword URIs identifies nodes that index all knowledge units tagged the referred keyword.

4.3.1 Basic Solution

This is a simple solution to build a distributed keyword index. It extracts the keyword concepts from knowledge units. Then the URI of the concept is used as a key to save the knowledge unit URI with. Over time different nodes build lists of knowledge units tagged with the keyword they are responsible for. Figure 4.2 illustrates how the indices are built.

To search for knowledge units with a keyword, the URI of the keyword concept has to be hashed. The resulting key can be used to fetch the list of knowledge units tagged with the keyword concept. More complex queries fetch multiple candidate-lists and process them locally to produce a final list. A search for knowledge units tagged with keyword x and y retrieves a candidate-list with *hash(x)* and another one with *hash(y)*. In the final result only those knowledge units appear, whose URIs are contained in both candidate-lists.

4.3.2 Advanced Solutions

This sections presents two more advanced approaches to facilitate keyword searching. SHARK departs from the idea of DHTs and organizes a different P2P topology for native support of rich keyword-based search. RDFCube on the other hand builds an index on top of a DHT-based distributed triple store. It can be used to answer complex queries for RDF data more efficiently.

SHARK – Symmetric redundant Hierarchy Adaption for Routing of Keywords

SHARK is a concept for search in P2P networks that is different from flooding search or DHTs. It arranges nodes in semantic clusters, so-called groups of interest (GoI) according to the objects they store and their request behaviour. It is specifically designed for rich keyword searching. Search requests can be specified in multiple dimensions. Requests are routed to the responsible GoIs according to all dimensions specified in the search. For example a search for music can be specified according to genre, decade, and name of the song. Each dimension can be sub-divided into further levels of granularity, for example sub-genres. SHARK provides rich keyword search, better than DHTs, that can not provide keyword search without additional structures. SHARK also outperforms flooding based search significantly and shows good scalability properties [20].

However SHARK is not intended to serve only for keyword indexing but defines the organisation of the entire P2P network. This means SHARK is not made to be combined with DHTs. Its search capabilities are tempting to be used in Knowledge Sharing Systems nevertheless. It would be worthwhile to investigate whether SHARK can be combined with knowledge organization ontologies, such as SKOS to enable multi-dimensional keyword search based on concept schemes. But this thesis focuses on a DHT based implementation and therefore it will not elaborate on these outlooks at this point.

RDFCube

RDFCube [17] is an indexing scheme that improves query processing efficiency of distributed triple stores, such as RDFPeers which is described in section 3.4.1. Many P2P based triple stores generate unnecessary traffic because in order to answer more complex queries they need to retrieve many candidate triples and then sort out those that match. If we have a query as follows (the syntax of the query is based on the SPARQL query Language for RDF [22]):

?x foaf:knows <http://example.org/person/John>
?x foaf:age "20"

?x denotes a variable. In this case a subject. Therefore we are interested in subjects that are 20 years old and know a specific person, John. Such queries are also called *join* queries.

In order to resolve join queries, a distributed triple store such as RDFPeers needs to retrieve all triples that match *?x foaf:knows <http://example.org/person/John>* from the DHT. If John has many friends, then this list may be big. Then all triples matching ?x foaf:age "20" would be retrieved. Again this might be a big list. But for the result we only want those triples that match both sub-queries. Therefore we need to gather all candidate triples locally and process them until we find all subjects that appear in both result sets. Most likely only a small fraction of subjects will match both queries. Therefore we transmitted a lot of data to resolve our query that we did not need in the end. With an appropriate index we can avoid transmitting triples that can not contain any results. RDFCube builds and maintains such an index.

RDFCube is a second DHT that stores only index information. In RDFCube the hashspace is three-dimensional and forms a cube. Depending on number of nodes the cube is split into cells for which different nodes are responsible. Triples that need to be saved get their subject, predicate, and object hashed. The resulting hash key consists of three numbers that represents a point in the three dimensional hashspace. The point is located within exactly one cell. This cell has the corresponding existence bit set to 1. This denotes that within this cell triples exist.

When a query such as the one described above is posed, it gets mapped into RDFCube. For *?x foaf:age "20"* candidate triples are located in the cell sequence [*, hash(foaf:age), hash("20")] (Note that two of the three dimensions are defined, one dimension is a variable). The result could be something like [*,5,1]. This means candidate answers may be in points [0,5,1], [1,5,1], [2,5,1]. For each of those coordinates the mapped cell is checked whether it has the existence bit set to 1. This result is a bit sequence like [1,0,1] (denoting that triples exist in [0,5,1] and [2,5,1]). By processing the second query in the same manner, another cell sequence of [*,2,3] and a bit sequence of [0,0,1] is derived. A *join* operation with both queries can be performed by applying the AND operation to both bit sequences. That results in [0,0,1]. Therefore we know that answers to our join query can only be located in cells [2,5,1] and [2,2,3]. This information can narrow down the amount triples that need to be retrieved.

Evaluations of RDFCube show that it reduces the transferred data size when resolving join queries significantly and the index construction is relatively inexpensive.

Other queries that fall into the same category are transitive queries like:

?x kss:tag ?y
?y dc:subject ?z
?z skos:prefLabel "Keyword"@en

Those queries we need to resolve for keyword searching. RDFCube is an ideal candidate to be used in connection with the RDFPeers triple store for project SciMantic.

4.4 The Knowledge Sharing System

The Knowledge Sharing System is the application that allows users to create, edit, and delete knowledge units and join or leave Knowledge Sharing Networks. A single KSS instance acts as a node in a P2P overlay network when it joins a KSN. To provide for this functionality a set of services with web interfaces are defined. For human interaction the KSS should provide a Graphical User Interface (GUI) in the form of a web site.

The architecture of the Knowledge Sharing System is partially defined by the Apache Clerezza semantic web platform. Clerezza is an OSGi based application. As a consequence the KSS services are deployed as OSGi bundles following the principle of service oriented architecture (SOA). OSGi subscribes to the SOA paradigm but it is not made specifically for web services. It is mainly focusing on achieving low coupling between services inside a Java Virtual Machine (JVM). Because SciMantic builds an application for the Semantic Web, web interfaces are required as well. Clerezza implements the JAX-WS API [2] for building RESTful web services (more details on REST in section 4.4.2).

4.4.1 KSS Services

The KSS consists of the three services described in this section. The services do not compose any hierarchy and may interact with each other trough programmatic OSGi interfaces. Web interfaces allow users to interact with these services. A user can be either a human (typically using a GUI) or a different application using the RESTful HTTP interface directly.

Knowledge Unit Sharing Service (KUSS)

This service joins and leaves knowledge unit networks and publishes knowledge units in them. For this purpose it uses a DHT service because knowledge sharing networks are

[2] `https://jax-ws.dev.java.net/`

implemented as P2P overlay networks. It must be able to participate in multiple networks at the same time, and also provides a subscription service that notifies users of changes to knowledge units defined by user criteria. It offers a RESTful web interface to publish knowledge units and for the subscription service.

Knowledge Unit Management Service (KUMS)

This service provides for local storage of knowledge units and offers a RESTful web interface for management operations including create, read, update, and delete (CRUD). Local storage can be accessed using Clerezza services.

Knowledge Unit Tagging and Searching Service (KUTSS)

This service offers RESTful web interfaces to tag knowledge units with appropriate keywords and to search for knowledge units tagged with specific keywords. Searching is not restricted to local storage.

4.4.2 KSS Interfaces

As KSS services are to be built on top of Clerezza, they can be used programmatically within the OSGi environment by any bundle running in Clerezza. OSGi allows to declare service components with Java interfaces. But in order to interact with KSS services across the application boundary, web interfaces are defined. The interfaces are designed according to the REST architectural style. Therefore the focus lies on using the general HTTP interface as it is specified rather than declaring entirely new interfaces.

RESTful HTTP interfaces

Each KSS service has a defined URL path that it listens on for HTTP requests. The root of this path is always the local Clerezza instance.

In compliance with REST [14] the different HTTP methods are used according to the way they are defined by IETF RFC 2616 [13]. In the SciMantic KSS the create, read, update, delete, share, subscribe, tag, and query operations are mapped to the HTTP methods as shown in table 4.1. Some argue that POST is used to create new resources and PUT is used to update them [23]. RFC2616 specifies that PUT can be used to create new resources and additionally:

> The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a

Operation	HTTP Method	Arguments
/kss/kuss/		
subscribe	POST	RDF Graph
/kss/kuss/ku/<id>/		
share	POST	share=<true|false>, ksn=<string>
/kss/kums/ku/<id>/		
create/update	PUT	RDF Graph
read	GET	none
delete	DELETE	none
/kss/kutss/		
query	POST	query=<string>
/kss/kutss/ku/<id>		
tag	PUT	RDF Graph

Table 4.1: RESTful KSS service interfaces and their arguments.

> separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request – the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource.

According to this definition we can not use the POST method to alter resources on the path specifying a knowledge unit, because the knowledge unit is the affected resource and not the data-processing resource. To further comply with REST, directory style URIs are used and the services are designed in a stateless manner.

The detailed interfaces of the services are:

share: Takes arguments that specify whether the knowledge unit should be shared or deleted from the specified KSN. Only the owner can share and delete knowledge units.

subscribe: Takes an RDF graph containing user-defined criteria for subscriptions to keywords or knowledge units.

create/update: Create and update are the same operation. In both cases an RDF graph containing a knowledge unit is uploaded to the specified URI. If a knowledge unit already exists at the URI, the old version is replaced if the user has the required privileges.

read: Returns the requested knowledge unit in the format specified by the HTTP accept header. If the requested format is unsupported an error is returned. Read also fetches knowledge units from the KSN if necessary.

delete: Deletes the requested knowledge unit if the user has the required privileges. Deleted knowledge units are removed from the KSN as well.

query: Executes the submitted query and returns a list of matching knowledge units. Search takes place locally and in the KSN.

tag: Tags the knowledge unit with the tags contained in the provided RDF graph.

Chapter 5

Implementation

This chapter describes the details of the KSS implementation that has been developed as part of this thesis. The reader is reminded that only parts of the concept laid out in section 4 have been implemented and that the implementation is of prototypical nature. This means it provides the minimal functionality required but it may not always have all the designed features. The implemented functionality is the following:

- A *DHT Service* running as an OSGi bundle that enables KSS instances to take part in P2P overlay networks.
- The *Knowledge Unit Sharing Service (KUSS)* has been partially implemented. It can connect to KSNs and share or delete knowledge unit in the KSN.
- The *Knowledge Unit Management Service (KUMS)* has been implemented completely. All the required operations specified in section are available.
- Only the searching and indexing functionality of the *Knowledge Unit Tagging and Searching Service (KUTSS)* has been implemented. The service can index new knowledge units in a distributed fashion and supports simple keyword-based search for knowledge units.

The following sections provide a documentation of the current implementation from a systems point of view and discusses deviations from the original concept. Furthermore implementation problems and their solutions are presented.

5.1 Architecture

Conceptionally this implementation consists of two major components, the Knowledge Sharing Network (KSN) and the Knowledge Sharing System (KSS). The KSN is a P2P overlay network implemented as a Chord DHT [9] that provides bootstrapping mechanisms for new nodes to join the network. The KSS is node participating in P2P overlay networks

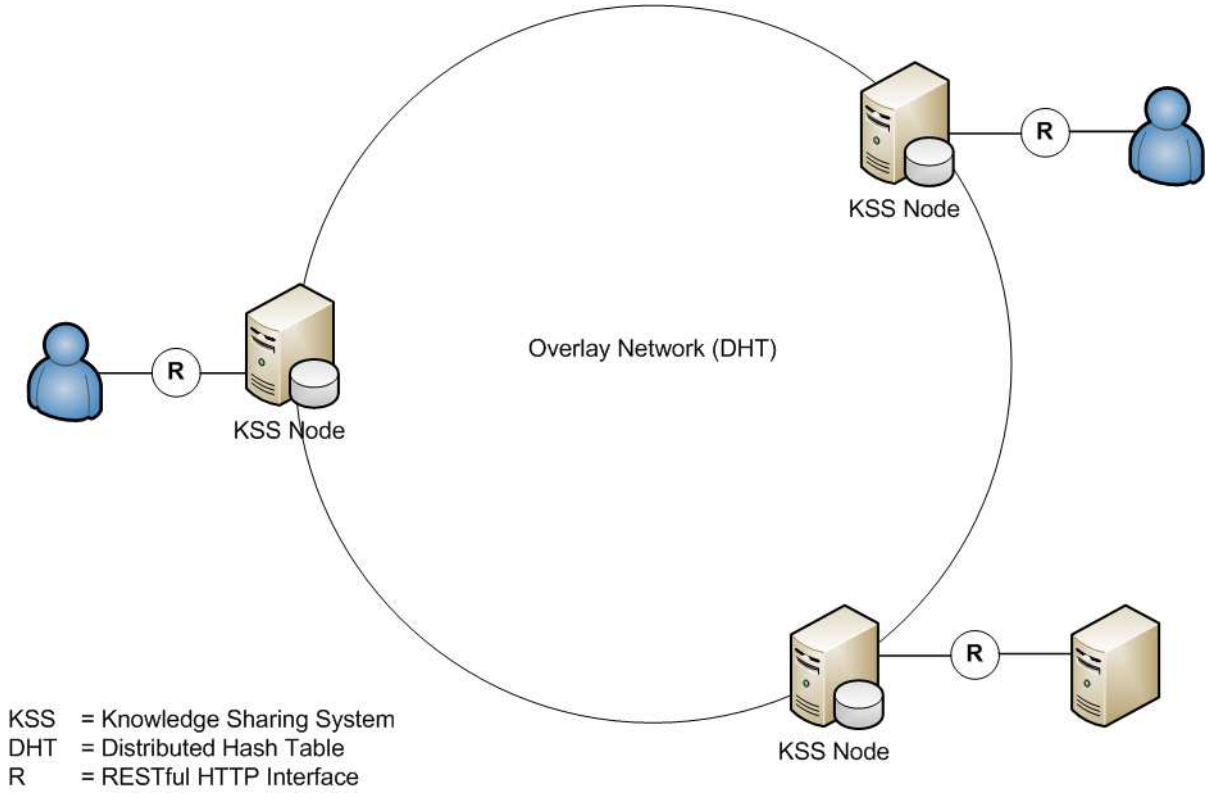

Figure 5.1: KSS Architecture Overview: Users can interact with nodes via a web interface. Nodes are connected trough a DHT but also offer local storage.

and a Semantic Web application built on top of Apache Clerezza[1]. The KSS can act as a stand alone application but sharing of knowledge units between KSS nodes requires the KSN. Figure 5.1 shows how both components are interconnected.

5.1.1 Knowledge Sharing Network

The Knowledge Sharing Network is implemented outside the Apache Clerezza platform. Currently it consist only of a Java command line application that sets up a bootstrapping server. Details on the specific DHT implementation chosen follow in section 5.3.1.

The KSN infrastructure can be set up by activating the OpenChord console. It is located in the *SciMantic: Utilities* Maven module. In its target directory issue the following commands to set up a bootstrap server on localhost, listening on port 8181. (note: for the target directory to appear the project must be built first):

```
$ java -jar knowledgesharing.utils-1.0-SNAPSHOT-jar-with-dependencies.jar\\
 occonsole

invoking class ch.uzh.csg.knowledgesharing.utils.OpenChordConsole
SciMantic OpenChord Console
```

[1] http://incubator.apache.org/clerezza/

```
-----------------------------------
Type 'help' for available commands.
-----------------------------------
create ocsocket://localhost:8181/

Creating overlay network with bootstrap URL ocsocket://localhost:8181/...
... overlay network successfully created.

stop
Shutting down overlay network...
... overlay network successfully shut down

exit
```

5.1.2 KSS Node

Figure 5.2 shows an overview of the software components of a KSS Node. The node itself is only an abstract concept. In practice it is an instance of the Apache Clerezza platform. Clerezza is a Java platform running inside an Apache Felix [2] OSGi container. The software components running inside an OSGi container are termed bundles. They can be installed and removed at runtime. Furthermore bundles can contain service components. Services implement Java interfaces and can be used by other bundles. Service Components can be activated and deactivated dynamically at runtime and react to other services becoming available. Additionally Clerezza offers non-OSGi libraries. We use the general term "module" to combine libraries, bundles, and service components. To extend Clerezza with KSS functionality new KSS modules have been implemented.

To run an instance of Clerezza build or download a launcher and run it:

```
$ java -jar \\
org.apache.clerezza.platform.launcher.sesame-0.5-incubating-SNAPSHOT.jar
```

5.2 KSS Services

Figure 5.3 shows an UML diagram of all the implemented OSGi services and dependencies between them. Thanks to the OSGi framework there are no strong dependencies between the services such that they can exist without the presence of the other services. The `P2PProvider` service interface is central to the DHT related functionality of the KSS. Without a `P2PProvider` instance being present, knowledge units can not be shared, indexed, or searched using keywords.

Services are available as Maven modules and they are deployed as OSGi bundles in Clerezza. Clerezza offers a user interface for installing new bundles at `http://localhost:`

[2] `http://felix.apache.org/`

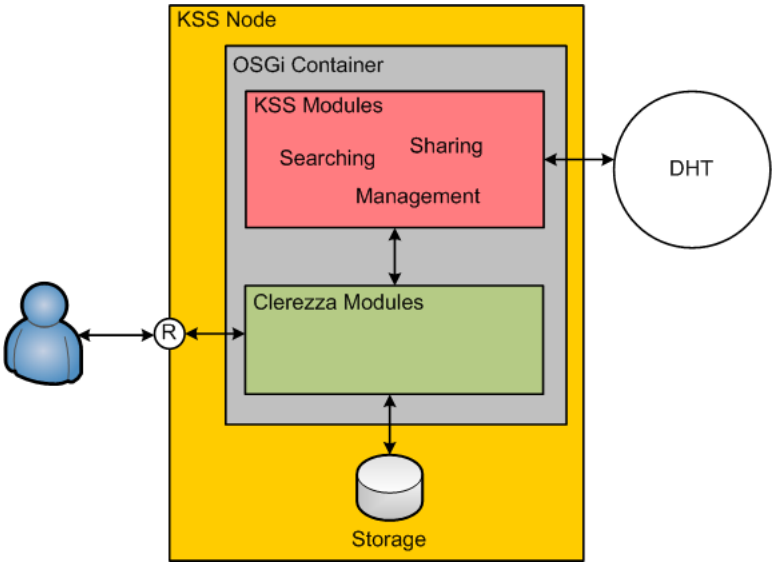

Figure 5.2: KSS Node Overview: KSS modules run inside an OSGi container and interact with Clerezza modules that offer web interfaces and local storage

`8080/user/admin/control-panel` (make sure to run a Clerezza instance first, login as user: admin pw: admin). All KSS services depends on the *SciMantic: KSS – Ontologies* bundle. Therefore it is recommended to install it first.

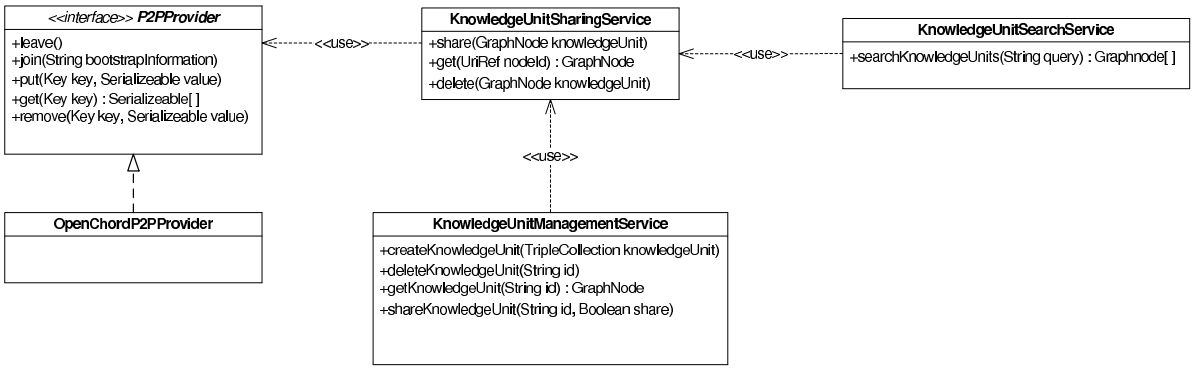

Figure 5.3: UML class diagram of the KSS Services and their dependencies. The P2PProvider interface hides the P2P service implementation from the rest of the system. Therefore the P2P service can be easily changed.

5.2.1 Knowledge Unit Sharing Service (KUSS)

The Knowledge Unit Sharing Service has two responsibilities. Joining and leaving knowledge sharing networks and sharing and deleting knowledge units in KSNs. While the service fully supports sharing and deleting knowledge units in a KSN, the KSN to join has to be statically configured for this service. Currently there are no programmatic interfaces to manage multiple KSN connections at the same time. There is no technical reason that would prevent it. This is just one of the compromises of this implementation being of prototypical nature. The connection is activated when the service is started. Then

the KSS joins a DHT using the specified bootstrap URL. When the service or the entire application is shut down again, the KSS leaves the DHT in an orderly manner.

No functionality to allow for a subscription service has been implemented. Such a service is of very complex nature, and its implementation is not required in order to achieve the goals of this thesis.

Another change from the original design is that no web service is offered. Again there is no technical reason for this. The *share* interface (c.f. table 4.1) has been moved to the management service in very early stages of the development and has remained there, because there is no pressing reason to move it back. This does not mean that the management service has taken over the functionality of the sharing service. Only the web interface has moved there.

The `KnowledgeUnitSharingService` can be found in the *SciMantic: KUSS – Knowledge Unit Sharing Service* Maven module. It has a dependency on `P2PProvider` and will not start without it being available. There are two utility classes provided in the same package, `SerializableGraph` (a RDF Graph that implements the Java Serializable interface) and `StringKeyImpl` (representing a string valued `P2PProvider` (hash-)`Key`).

5.2.2 Knowledge Unit Management Service (KUMS)

This service has been completely implemented. It supports creating, editing, retrieving, and deleting of knowledge units. New knowledge units are created in local storage. When the knowledge unit is shared, it is retained in local storage but a property, that specifies that it is shared is added. Here we deviated from the original design where this property should denote the KSN it is shared in. This is a consequence of the current implementation, which only supports the connection with a single KSN.

The web interface of this service have been implemented largely according to the specification summarized in table 4.1. The *create* operation is mapped to the PUT HTTP method and expects a serialized RDF graph to be submitted in the request body. The submitted graph is then parsed and searched for nodes of the *rdf:type kss:KnowledgeUnit*. These nodes are extracted using depth-first graph traversal with loop detection and saved in local storage. In order to edit a knowledge unit one can use the same PUT method again and submit an existing knowledge unit. If the user uploading the new graph matches the creator of the old knowledge unit, the knowledge unit is overwritten with the new one. Edit functionality is therefore given by first retrieving a knowledge unit, editing it, and uploading it again. As explained in section 5.2.1 the *share* interface belonging to the KUSS has been moved to this service.

The `KnowledgeUnitManagementService` can be found in the *SciMantic: KUMS – Knowledge Unit Management Service* Maven module. It depends on `KnowledgeUnitSharingService` `KnowledgeUnitSearchingService`. The user may interact with its web services in the following manner (interaction is shown using the CURL [3] commant line tool):

[3] `http://curl.haxx.se/`

For any command the the option `-b "auth=YWRtaW46YWRtaW4="` must be added in order to authenticate as the *admin* user. For knowledge unit URI of `http://scimantic.ch/2010/09/kss/ku/8c5cbfe1-702c-4546-8543-bcca8738c6f0`, the <*id*> is the last part: `8c5cbfe1-702c-4546-8543-bcca8738c6f0`

create/update: *graph.ttl* contains a serialized RDF graph, it is necessary to provide the correct content-type.

```
curl -T graph.ttl -H "Content-Type: text/turtle" \\
"http://localhost:8080/kss/kums/ku/<id>"
```

read:

```
curl -X GET "http://localhost:8080/kss/kums/ku/<id>"
```

delete:

```
curl -X DELETE "http://localhost:8080/kss/kums/ku/<id>"
```

share:

```
curl -X POST "http://localhost:8080/kss/kums/?id=<id>&share=<true|false>"
```

5.2.3 Knowledge Unit Tagging and Searching Service (KUTSS)

This service implements functionality to index knowledge units and enables keyword-based search. The indexing is implemented according to the basic solution defined in section 4.3.1 and is done in the same DHT, in which the knowledge units are saved. Indexing happens automatically when a knowledge unit is shared. The exact reasons for choosing the basic solution will be discussed in section 5.3.1. The searching service implements a very basic query parser. Queries are strings with space separated keywords. The string is interpreted as a conjunction of keywords and therefore only results that match all keywords are returned. The results are returned as a list of knowledge units.

The web interface of this service offers the *query* (see table 4.1) operation only.

No tagging related functionality has been implemented because it is outside the scope of this thesis.

The `KnowledgeUnitSearchingService` can be found in the *SciMantic: KUTSS – Knowledge Unit Tagging and Searching Service* Maven module and it depends on a `P2PProvider`. The user may interact with its web services in the following manner:

query:

```
curl -X POST "http://localhost:8080/kss/kutss?query=World%20Hello"
```

5.3 DHT Service

The `P2PProvider` interface defines a very basic DHT interface that can be fulfilled by any DHT service. DHT services implement this interface. The KSS should not depend on a concrete DHT service because it may become necessary to exchange the DHT service in the future.

`OpenChordP2PProvider` is an OSGi service that implements the `P2PProvider` interface and offers a DHT API using the OpenChord DHT implementation [4]. Therefore any component that uses a `P2PProvider` will in reality use `OpenChordP2PProvider`. Multiple `P2PProvider`s can coexist. Currently there is no mechanisms to prefer one implementation over the other at runtime but such functionality can be implemented.

This section describes why the OpenChord DHT implementation has been chosen for the KSS, the problems associated with adapting such implementations for OSGi, and the consequences from the chosen implementation.

5.3.1 DHT Implementation

The choice of a DHT implementation was very problematic. Originally JXTA [5], respectively it's Java implementation JXSA has been chosen because it appeared to be one of the most mature DHT implementations available in Java, and it offered many desirable features. Initial testing with the framework was successful but the implementation of an OSGi bundle was not possible because of missing dependencies. After many attempts to package all dependencies into OSGi bundles this idea was abandoned when sun implementation classes (from the packages com.sun.* or sun.*) were still missing. It turned out that some of the dependencies directly referenced these classes. But those are classes of the Sun JVM and and the OSGi container did not export them. We found a way to expose these dependencies by editing the configuration of the Apache Felix OSGi container and manually exporting these classes. But that would create a dependency on the Sun JVM. Any other valid JVM implementation would be incompatible and KSS instances would require manual configuration.

The same dependency problems appeared when trying to run other DHT implementations (BambooDHT [6] and FreePastry [7]) in Apache Felix. Only OpenChord could be successfully implemented as an OSGi bundle. Since these problems caused a substantial delay we did not try to implement any other DHT implementations any more. OpenChord turned out to offer all the basic functionality needed but not more.

The `OpenChordP2PProvider` service is located in the *SciMantic: KUSS – Knowledge Unit Sharing Service* Maven module. It depends on the *SciMantic Ext: OpenChord* bundle that exports the OpenChord implementation.

[4] `http://open-chord.sourceforge.net/`
[5] `http://jxta.kenai.com/`
[6] `http://bamboo-dht.org/`
[7] `http://www.freepastry.org/FreePastry/`

5.3.2 DHT Organisation

Because there are no mechanisms in place to manage multiple DHT connections it has been decided to store the knowledge units and the distributed keyword index in the same DHT. Following the example of RDFCube (see section 4.3.2) it would be more elegant to create a dedicated DHT for indexing. The DHT organisation is depicted in figure 4.2it is an exact implementation of the basic solution described in section 4.3.1. The storage of knowledge units has been implemented as an unoptimised RDFPeers (see section 3.4.1) implementation without query support. It became quickly became obvious that this solution exhibited bad performance (see section 6.1.2 for a performance analysis). This could only be optimized at the level of the DHT implementation but that is beyond the scope of this thesis.

In order to achieve good performance it has been decided to use the DHT not as a true distributed triple store but rather as a distributed knowledge unit store. Instead of saving each triple into the DHT, the knowledge unit is serialized and saved using a single key. That increased the system performance significantly.

5.4 Security and Access Control

Security in the current implementation can only be offered within the Apache Clerezza framework. Clerezza supports user authentication for web services and OSGi bundles. The Open Chord DHT implementation provides no security mechanisms on the other hand. The integrity of data in the DHT can never be guaranteed. Anyone can theoretically write into the DHT. For reading it is not a concern as anyone should be able to read. But in the case of editing existing knowledge units, it represents a problem. A custom OpenChord application can overwrite or delete anything from the DHT if it knows the bootstrap server address. Again the problem lies at the level of the OpenChord implementation that is inherently insecure.

Chapter 6

Evaluation

This chapter evaluates the implemented KSS. It aims to identify performance bottlenecks, discuss implementation aspects that affect performance in the application and to demonstrate how the application scales.

6.1 Considerations

The KSS services (KUSS, KUMS, and KUTSS) are not computationally intensive and are not expected to become bottlenecks of the implementation. Performance measurements show that the overhead added by the knowledge unit sharing service when retrieving knowledge units from the DHT can be neglected. It is indistinguishable from the variability of the data set.

The areas that deserve most attention in this evaluation are the web service implementation, the DHT service, and the DHT organisation. The web service is the interface to the outside and difficult to control. If a web service is popular, it will require considerable resources on the server to serve all clients. There are not many possibilities to avoid this completely but the service can be designed with efficiency in mind. The DHT on the other hand is the most complex component of the KSS solution. When data is entered it gets transmitted over a network to other computers. The task of routing to the desired computers in the overlay network and then to establish a connection for data transfer produces delay times that differ in magnitudes from local processing times. The manner in which the data is organized in the DHT directly affects the number of interactions the DHT service needs to initiate with other nodes. The expectation is that this is the most likely bottleneck.

6.1.1 Performance Analysis of Web Interfaces

Because the web services are implemented according to the REST architectural style we can expect them to fare well in terms of performance. RESTful web services are stateless

and tend scale well as a result. Stateless services relieve the server from keeping state information for every connection with a client. Keeping state information for a single connection is not necessarily complex but the effort scales proportionally to the number of connections. Most clients have sufficient processing power today. To prevent the server from becoming a bottleneck it is beneficial to let the client deal with state information where possible. The server can concentrate on providing the web services and has as many resources as possibly available for that task.

The KSS services follow these guidelines. Therefore functionality like editing and creating knowledge units is left completely to the clients. The KSS only validates the user credentials and submitted graphs. Accepted graphs are annotated with administrative information before they are stored. The KSS services are not concerned with the creation of the content of a knowledge unit. Only with its administration.

6.1.2 Performance Evaluation of a Distributed Triple Store

During development the idea to implement a true distributed RDF triple store has been abandoned for performance reasons. Section 3.4.1 describes that RDFPeers is an ideal candidate for implementation of the DHT service of the KSS. RDFPeers saves each triple of an RDF graph three times in the DHT. A very simple knowledge unit with content consists of at least 80 triples (an average knowledge unit is very likely to consist of at least 200 triples). In order to save a knowledge unit in an RDFPeers implementation, three times that many triples have to be saved. This amounts to about 240 triples for a simple knowledge unit. Assuming a complexity of *O(log N)* routing hops [8] for N nodes and a network of 100 nodes, for each triple two nodes are contacted. Therefore we produce 480 connections in the transport network to save a minimalistic knowledge unit. These connections were found to be a performance bottleneck because they cause high delays. Therefore it will be used as a measure of performance in the following discussion.

Retrieving knowledge units from an RDFPeer inspired implementation presents another performance problem. The naïve assumption is that if a knowledge unit URI is known, it can be retrieved with 160 connections, provided the assumptions used for insertion are valid. 80 triples need to be downloaded and for each triple two nodes are contacted. But this is only true when there is only a single knowledge unit in the DHT. When many knowledge units exist more triples need to be transmitted because knowledge units contain blank nodes. Blank nodes are indistinguishable among each other. Section 4.1.2 describes how the tagging properties point to blank nodes. Assuming an unoptimized algorithm, the search for he knowledge units with a specific keyword, *Moon*, looks like this:

?x kss:tag ?y
?y dc:subject ?z
?z skos:prefLabel "Moon"@en

We are interested in all *?x* fulfilling this query. Therefore we have to decompose the query into parts. First *?z skos:prefLabel "Moon"@en* is executed, z are the keyword concepts (see section 4.1.1). Then *?y dc:subject ?z* is executed for all z fetched before. But y are blank

nodes. The sub-query *?x kss:tag ?y* means: All knowledge units x tagged with anything. The blank nodes y fetched before, can not be distinguished among other blank nodes. All knowledge units that are tagged, even when the tag is a completely different one, are valid results and the corresponding triples are fetched. The problem is that there is a transitive relation $x \to y, y \to z \models x \to z$. It can not be understood when the involved triples are saved at different locations. First all possible triples need to be fetched from the DHT to establish this relation.

It is evident that the underlying DHT implementation must offer some optimizations, in order to enable a distributed triple store to perform well in such scenarios. The OpenChord implementation used in the KSS does not offer any optimizations for these scenarios, and therefore a different solution had to be found (see section 5.3.2). A possible optimization using RDFCube has been presented in section 4.3.2;

6.1.3 Performance Evaluation of the KSS DHT Service

The implemented KSS performs very well for keyword-search and knowledge unit insertion and retrieval. But it is not a true triple store. It exploits the fact that knowledge units can be identified by their globally unique URI and uses the URI as a key to save a serialized representation of the entire knowledge unit graph in the DHT. Assuming the numbers in section 6.1.2, it allows insertion of a knowledge unit with 2 connections, regardless of its size. Retrieval uses 2 connections as well. Keyword search can be done with 2 connections per keyword in the query and 2 connections to fetch each matching knowledge unit. For a knowledge unit tagged with 5 keywords this means 12 connections to find and fetch it.

But there is a performance bottleneck in the DHT service because it works synchronously. It only opens a single connection at a time and waits for it to be completed before it opens a new connection. OpenChord supports asynchronous interaction with the DHT but the KSS implementation has not implemented this so far.

6.2 Scalability Evaluation

In order to assess the scalability of the KSS, the evaluation focuses on measuring the change of the duration of specific tasks. Either the number of knowledge units or the number of nodes in the DHT grows during the scenario. The absolute duration value is not relevant to the evaluation. The evaluation is interested in its growth only. The tasks are insertion of data into the DHT and fetching of data from the DHT. Knowledge node insertion, keyword-search, and knowledge unit indexing is not differentiated because from the point of view of the DHT service, the operations are equal in procedure and connections required. Only the amount of data transferred is different.

The evaluations were run on a single physical node. OpenChord offers a protocol that allows a DHT to communicate inside a single JVM, bypassing the network interface. This has been used because it allows great flexibility in number of nodes and the results are

not affected by network transmission time. The DHT mechanisms are identical to using the TCP sockets protocol otherwise.

6.2.1 Scalability in terms of Knowledge Units

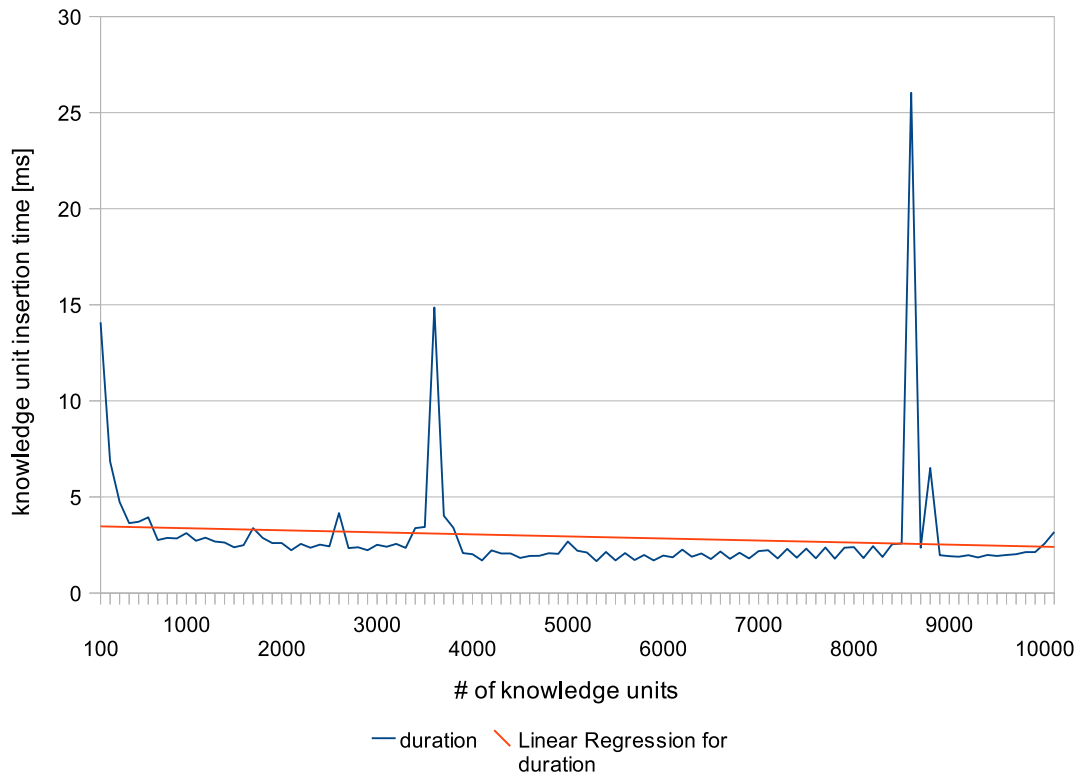

Figure 6.1: Average time to insert a new knowledge unit into the DHT in terms of knowledge units already present in the DHT.

In these scenarios the routing hops remain constant. The DHT has 10 nodes which suggests that on average one routing hop will be sufficient. The evaluations have been run with 100 and 1000 nodes as well but the graphs appeared very similar. The time measured is the average time needed to insert a knowledge unit. The graph shows how the DHT behaves when more and more knowledge units are saved in it. The distribution of the knowledge units over the DHT nodes is uniform. Figures 6.1 and 6.2 show a surprising image. The expectation was that the DHT becomes slower when it fills up but the trend lines show that it actually becomes faster. The reason for this is not clear. It might be an unrelated side effect (e.g. optimization of memory access by the operating system). Nevertheless it shows that the DHT is not becoming slower from being filled up (note that a single node has to save up to 1000 knowledge units in this scenario). Judging from the data either the DHT performance remains constant – which is what we suspect – or it becomes linearly faster indeed.

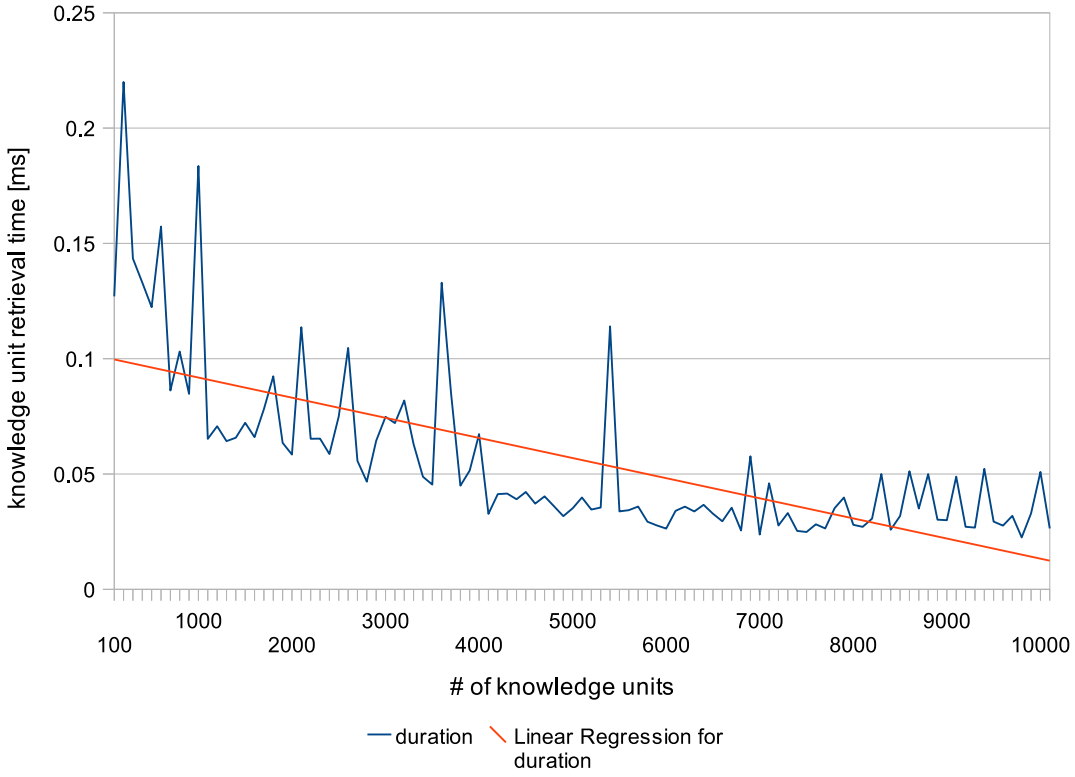

Figure 6.2: Average time to retrieve a new knowledge unit from the DHT in terms of knowledge units already present in the DHT.

6.2.2 Scalability in terms of Nodes

Theoretically all Chord DHT operations should scale logarithmically to the number of nodes [9]. Figures 6.4 and 6.3 show that the average time to retrieve and insert a knowledge unit is growing with a growing number of nodes. The trend line seems to suggest that the growth may be logarithmic. But it is not entirely clear. Evaluations with more nodes and data points would be helpful to decide whether the growth is linear or logarithmic but the available test environment can not deal with larger numbers. A scale of 10 to 1000 nodes seems to be a realistic measure for KSN membership in a useful real life scenario nevertheless.

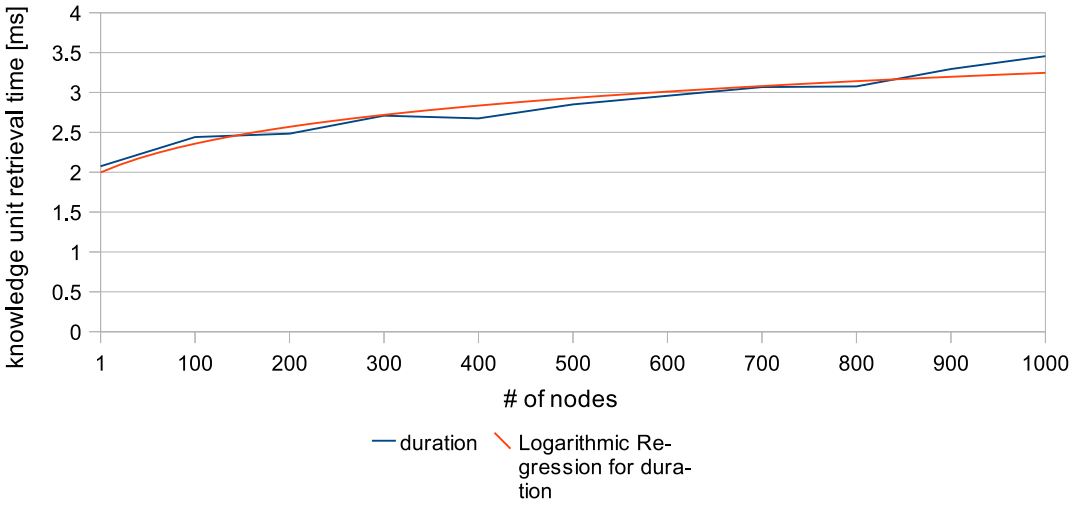

Figure 6.3: Average time to insert a knowledge unit into the DHT in terms of number of nodes in the DHT.

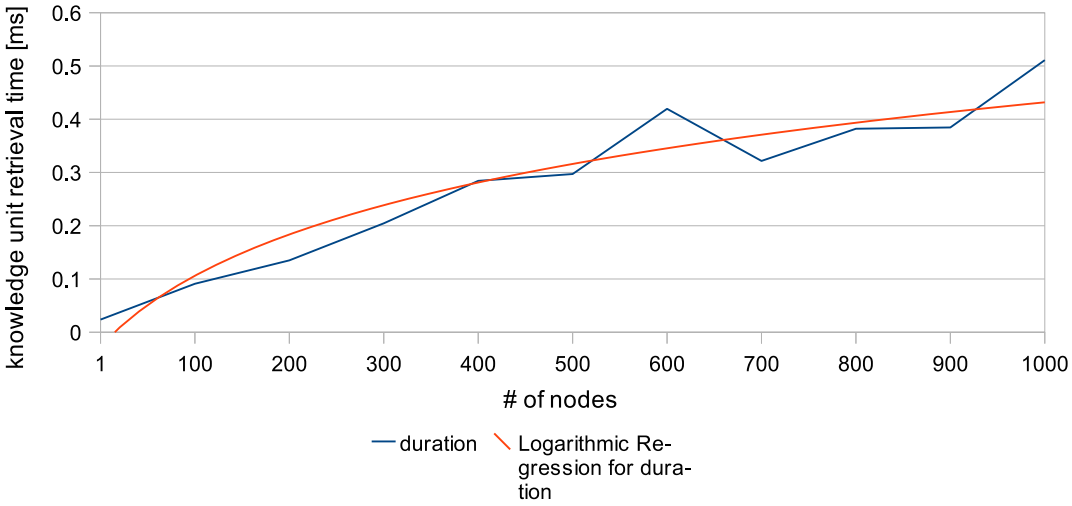

Figure 6.4: Average time to retrieve a knowledge unit from the DHT in terms of number of nodes in the DHT.

Chapter 7

Summary and Conclusions

In this thesis we first introduced two related knowledge management systems. The comparison with the SciMantic KSS shows that we share features with both implementation but that we also have a unique approach. Most notably we a DHT to share knowledge in. The other systems don't use DHTs. Also the open sharing approach (anyone who is interested can join and share knowledge units in a KSN) is unique. Sharing scenarios in the other systems are more localized inside organisations or by proximity of mobile peers.

In chapter 3 we briefly introduced theoretical concepts of knowledge management and knowledge management systems before the key technologies for this thesis have been presented. They are the Semantic and Peer-to-Peer networks. We concluded that chapter by looking on the combination of P2P and RDF in the form of distributed RDF stores.

With the theoretical background clarified we started to conceptually design the architecture of a knowledge sharing system that combines the advantages of Semantic Web technologies and P2P applications. Thanks to semantic web technologies knowledge units are not merely data any more but they can be assigned meaning that can be understood by machines and humans alike. Keyword-searching demonstrates how the Semantic Web can enhance P2P technology. RDF enables us to search for keywords with a well defined meaning as opposed to searching for strings that can have ambivalent meanings. And with P2P technology we can make this knowledge accessible by sharing it in an efficient and scalable manner. On the other hand P2P technology helps us to connect distributed knowledge repositories. This advances the linked data vision of the W3C.

In chapter 5 we described a prototypical implementation of the designed KSS. We learned that some things that we did not think about when designing the system can decisively influence the functionality of the system. We made that experience when trying to adapt a DHT implementation for OSGi. The problems we faced forced us to compromise on security and access control. We also learned that solutions that appear good in theory do not always work in practice, when we attempted to implement a true distributed triple store without optimizing the DHT service.

Nevertheless the final result fulfils the requirements of the thesis. All the required functionality has been implemented and a thorough evaluation of its scalability and performance

justifies the approach and implementation choices made. The advantages of combining Semantic Web technology and a DHT have been demonstrated by enabling semantically enhanced keyword search in a DHT. The disadvantages we experienced with the realisation that distributed RDF querying is very complex to implement in DHTs and that the poor capabilities of the OpenChord DHT implementation are responsible for many shortcomings of the current implementation.

7.1 Future Work

There are three areas that require further attention. First and foremost it is necessary to optimize the DHT Service. This may allow the implementation of a true distributed triple store with acceptable performance for complex queries.

Secondly security aspects need to be considered and implemented on the DHT side before a real world KSS can be implemented. Currently the DHT offers no authentication and authorisation at all.

And finally the KSS needs a graphical user interface in the form of a web site for better user interaction.

Bibliography

[1] Jena TDB Wiki. http://openjena.org/wiki/TDB.

[2] Mulgara Semantic Store. http://mulgara.org/.

[3] FP7 in Brief. http://ec.europa.eu/research/fp7/pdf/fp7-inbrief_en.pdf, 2007.

[4] RTD Guide. K-NET – Services for Context Sensitive Enhancing of Knowledge in Networked Enterprises, 2008.

[5] Maryam Alavi and Dorothy E. Leidner. Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS Quarterly*, 25(1):107–136, 2001.

[6] S. Bechhofer and A. Miles. SKOS simple knowledge organization system reference. W3C recommendation, W3C, August 2009. http://www.w3.org/TR/2009/REC-skos-reference-20090818/.

[7] DCMI Usage Board. DCMI Metadata Terms. DCMI recommendation, DCMI, January 2008. http://dublincore.org/documents/dcmi-terms/.

[8] M. Cai and M. Frank. RDFPeers: A Scaleable Distributed RDF Repository based on A Structured Peer-to-Peer Network. Technical report, Information Sciences Institute, Marina del Rey, CA, 2003. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.6822&rep=rep1&type=pdf.

[9] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, 2003.

[10] DERI (Ed.). Deliverable D1.1; K-NET Framework and Atate-of-the-Art Analysis. K-NET – Services for Context Sensitive Enhancing of Knowledge in Networked Enterprises, 2007.

[11] Hasan (Ed.). Deliverable D2.1; Functional Requirements and Analysis of Mechanisms for a Semantic Content Infrastructure. The SciMantic Project, 2010.

[12] UNINOVA (Ed.). Deliverable D1.4; K-NET Public Concept. K-NET – Services for Context Sensitive Enhancing of Knowledge in Networked Enterprises, 2008.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

[14] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[15] Apache Software Foundation. Apache clerezza. `http://incubator.apache.org/clerezza/`.

[16] M. Krötzsch, P.F. Patel-Schneider, S. Rudolph, P. Hitzler, and B. Parsia. OWL 2 web ontology language primer. Technical report, W3C, October 2009. `http://www.w3.org/TR/2009/REC-owl2-primer-20091027/`.

[17] A. Matono. Query Processing for Distributed RDF Databases Using a Three-dimensional Hash Index, 2006. `http://www.nesc.ac.uk/action/esi/download.cfm?index=3141`.

[18] R. McAdam and S. McCreedy. A critique of knowledge management: using a social constructionist model. *New Technology, Work and Employment*, 15(2):155–168, 2000.

[19] E. Miller and F. Manola. RDF primer. W3C recommendation, W3C, February 2004. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[20] J. Mischke and B. Stiller. Rich and scalable peer-to-peer search with SHARK. Technical report, 2003. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.8050&rep=rep1&type=pdf`.

[21] The SciMantic Project. `http://www.csg.uzh.ch/research/scimantic/`.

[22] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`.

[23] A. Rodriguez. RESTful Web services: The basics. `http://www.ibm.com/developerworks/webservices/library/ws-restful/`, nov 2008.

[24] T. Schwotzer. *Ein Peer-to-Peer Knowledge Management System basierend auf Topic Maps zur Unterstützung von Wissensflüssen*. PhD thesis, Technische Universität Berlin, 2006.

[25] A. Seaborne. RDQL – A Query Language for RDF. W3C member submission, W3C, January 2004. `http://www.w3.org/Submission/RDQL/`.

[26] The OSGi Alliance. OSGi service platform core specification, release 4.2. `http://www.osgi.org/Specifications/HomePage`, 2009.

List of Figures

List of Tables